IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

SPECIFICATION
Accompanying

Application for Grant of U.S. Letters Patent

TITLE: OBJECT ORIENTED ADN AND METHOD OF CONVERTING A NON-OBJECT ORIENTED COMPUTER LANGUAGE TO AN OBJECT ORIENTED COMPUTER LANGUAGE

CROSS-REFERENCE TO RELATED APPLICATIONS

Not Applicable.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

Not Applicable.

REFERENCE TO COMPUTER PROGRAM LISTING APPENDIX SUBMITTED ON COMPACT DISC

A Computer Program Listing Appendix submitted on Compact Disc is included and the material contained on the Compact Discs is hereby incorporated by reference.

Copies 1 and 2 of the discs include the following:

Name	Size	Type	Last Modified
adn30.l	26 KB	L File	12/30/2000
adn30.y	99 KB	Y File	12/30/2000
ADNinterpreter30.sim	517 KB	SIM File	12/30/2000
ADNparser30.sim	411 KB	SIM File	12/30/2000
Dbms	72 KB	Text Doc.	12/30/2000
Server	20 KB	Text Doc	12/30/2000
Software_Util30.sim	163 KB	SIM File	12/30/2000
user_extensions	5 KB	Text Doc.	12/30/2000
Utilities	56 KB	Text Doc.	12/30/2000

FIELD OF THE INVENTION

The field of the invention relates to converting a non-object oriented computer language to an object-oriented computer language more particularly, the invention relates

to converting a non-object oriented computer language to an object-oriented computer language while maintaining existing code structure and data structures.

BACKGROUND

Computer languages have evolved over time as computer systems have become more complex and as instruction execution speed has increased. Society is becoming more dependent on computer systems and advance computer-programming languages. However, before the last decade, most computer programs were written in non-object oriented languages. The non-object oriented computer languages implemented simple logic statements which allowed the following: basic data input and output operations, implementation of subroutines which could be called and returned from, and the focus of the programmer to be placed on the procedures of the language or the program. Within the last several years however, a paradigm shift has occurred toward programming in an object-oriented language. In an object-oriented language, the programmer focuses on the data in the program and the methods that manipulate that data rather than focusing on the procedures of the language. Object-oriented languages are usually easier to understand. Examples of object-oriented language include C++ and Java.

In an object-oriented system, you solve your problem in terms of objects which occur in the context of the problem and objects are almost everything in common object-oriented systems. Objects allow you to define entities relevant to your particular program rather than strictly expressing your solution to a problem essentially in terms of characters and numbers as is required by non-object oriented language.

A class is a term used to describe a specification for a collection of objects with common properties. A class is also a collection of data and methods that operate on that

data. The data and methods describe the behavior and state of an object. Classes are hierarchical, that is subclasses inherit behavior from the classes above it. A class describes the requirements for a collection of objects and may be thought of as a template which defines what makes up the particular object. A class definition of an object lists all the parameters that the programmer needs to define the object of that particular class. Instance variables or attributes of a class are commonly used to define these parameters. Objects can include the methods that operate on it as well as the data that defines the object. This allows for ease in programming. Typically, object-oriented programs take longer to design than non-object oriented programs, as care must be taken to design the classes that will be necessary for your program, however, object-oriented programs are also much easier to maintain and expand.

However, many businesses have spent considerable time, effort and money in developing computer programs to control all aspects of their organizations. Many of these programs were developed using standard programming techniques that were available prior to the development of object-oriented design. As such, businesses must consider the time necessary and the cost associated with modifying or replacing existing programs with easier to maintain object-oriented programs. Therefore, any advancement in the ability to convert a non-object oriented language program to an object-oriented language program would be advantageous.

SUMMARY OF THE INVENTION

Object Oriented ADN and a method for converting a non-object oriented language to an object oriented language is shown. First, an existing object oriented language must be selected. An object oriented language is selected to provide for ease of

translation with regard to syntax and grammar. Second, the non-object oriented language is selected. This is the language the programmer desires to be converted to an object oriented language. The requirements for the new object oriented language are then defined. The requirements can be expressed in a document including a set of enhancements to be made to the existing language. Next, the specific syntax and grammar are selected. The object oriented extensions are then developed. The object oriented extensions allow for the existing language and data structure to be developed coextensive in the object oriented environment. Programs written in the existing language are still executable in the new object oriented language. Finally, the new object oriented language is prepared based upon the criteria outlined previously.

The Object Oriented ADN including an application logic function, data types and scopes, a class for message instancing, client workload models, server process infrastructure, database models, operating system models, statistics capability, utility classes, and garbage collection.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained when the following detailed description of one exemplary embodiment is considered in conjunction with the following drawings, in which:

FIGURES 1A-1B are flow diagrams of the conversion to object oriented environments process;

FIGURE 2 is an exemplary embodiment of the existing ADN language code; and

FIGURE 3 is an exemplary embodiment of the new object oriented ADN language code.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

In the description which follows, like parts are marked throughout the specification and drawings with the same reference numerals, respectively. The drawing figures are not necessarily drawn to scale and certain figures may be shown in exaggerated or generalized form in the interest of clarity and conciseness.

FIGURES 1A-1B illustrate a method of converting a non-object oriented computer environment to a new object-oriented computer environment. The process begins with Start 100 on Figure 1A. Next, in Step 102, the existing object-oriented computer environment must be identified. An existing object-oriented computer environment would include commercially available object-oriented languages such as Java, which is provided by Sun Microsystems Inc. Typical benefits of a commercially available object-oriented computer language include improved application extendibility, maintainability, data hiding and encapsulation which makes code reuse easier, and the ability to allow the user additional power and flexibility to implement complex applications. Further, if Java is selected, the automatic garbage collection feature and associated object reference counting design is particularly useful in certain languages, especially simulation language. Java's syntax is also based on C++ language syntax. Java's selection may also be appropriate if the legacy language is also based on C or C++ language syntax. Next, in Step 104, the non-object oriented computer environment is identified. The non-object oriented computer environment includes languages which

implement structure other than the object-oriented methodology and are desired to be converted to an object oriented system. Thus, older existing languages which pre-date the use of object-oriented analysis and design are potential candidates for this conversion. In one embodiment, the Application Definition Notation (ADN) language would fall in this category as ADN is a special purpose programming language for scripting the behaviors of clients and servers and is not written in an object oriented form. ADN was especially designed for use with a simulation tool and includes the features to handle concurrency, message passing, simulation time management, and simulated resource usage. The ADN features include C-like assignment in control flow statements, variables having different scopes and lifetimes, concurrent programming using threads and processes, simulation control and resource usage statements, input/output statements and a wide variety of built in functions. Next the requirements for the new object-oriented computer environment must be defined in Step 106. A requirements document can be produced from a set of enhancement requests and an understanding of the non-object oriented computer environment. Further, basic requirements such as reusability would factor in the determination of defining the requirements. The requirements document can be used to identify any future functionality or improved functionality of the non-object oriented computer environment. Also, the requirements document can identify functionality present in the non-object oriented computer environment which will be advanced through the modification or conversion to the object-oriented computer environment. Next in Step 108, compatible grammar and syntax is selected. The grammar and syntax of the new object-oriented computer environment must be compatible with the non-object oriented computer environment. The compatibility is

necessary as the non-object oriented computer environment language is to be extended to provide the new object-oriented capabilities. The beginning of the selection of the grammar and syntax occurs in Step 102 during the identification of an existing object-oriented computer environment. The existing object-oriented computer environment should be selected so that a standard object-oriented language specification is available and can be used as a reference document. Within this reference document, the use of the semantics of the reference language can be used as a guide to ensure that the functionality is implemented as part of the object-oriented conversion. Further, when the syntax or grammar is not exact, then the non-object oriented computer environment syntax should be followed as closely as possible to minimize the effort in determining the new object-oriented capabilities. Any language key words not present in both the existing object-oriented computer environment and the non-object oriented computer environment should be reserved for use in subsequent implementation. Next, the object-oriented extensions are developed in Step 110. The object-oriented extension design should satisfy several goals. First, the object oriented computer environment should not burden the legacy system user in cases where the new OO functionality is not used. The interface to the new OO features should appear natural and easy to learn for the legacy system user. Second, the object oriented features of the new object oriented computer environment should be able to access the non-object oriented computer environment information so that existing applications can take full advantage of the new object oriented functionality. The non-object oriented computer environment application integrity should be preserved through the internal creation and management of key structures including structures as processes, threads and/or messages.

In one embodiment, to preserve the integrity of the legacy system while giving the user more control, an object implementation was designed to include a header structure and a data structure. The object header structure was defined for use by all objects in the design. The mapping of all user-defined information is made to secondary structure. The object header points to either a user defined structure or a non-object oriented computer environment data structure. By this approach, the non-object oriented computer environment information can be accessed in the same way as other object-oriented information. Second the object reference type was added to the semantics of the non-object oriented computer environment generic variable so that it was able to access any type of information in the new object-oriented computer environment. Next in Step 112 on Figure 1B, the general purpose utility classes were developed. No general purpose utility classes existed in the non-object oriented computer environment. Therefore, to drive the full functionality of object-oriented operations, the general purpose utility classes must be added. Next in Step 114 the new object-oriented computer environment is prepared. In preparing the new object-oriented computer environment, the requirements defined, the grammar and syntax selected, and object-oriented extensions must be considered to implement and develop the new object-oriented computer environment to accomplish the goals of completeness and ease of use. These steps of preparing the new object-oriented computer environment and in particular, the actual code generation, is not listed herein for brevity as various methods are available for code generation once the requirement specifications, grammar and syntax selected, and object-oriented extensions are developed. Any of various common code generation techniques

are usable and their use does not detract from the spirit of the invention. The method ends in Step 116.

Referring now to Figures 2 and 3, examples of the ADN code and object-oriented ADN code are shown respectively. Figure 2 includes a portion of the ADN code which was prepared prior to the conversion to the object-oriented ADN. The file shown is the system.ADN. Figure 3 shows the object-oriented ADN code for the system.ADN after conversion to the object-oriented ADN. Through the method described previously, the ADN code was transformed to object-oriented ADN. The object-oriented ADN however, still possesses many of the same parameters and data structures contained in the original ADN. For example, on page 2 of Figures 2 and 3, the variable constants INITIALIZEsvc 202 and 302 are set to zero in both figures. However, on page-7 of Figure 3, the object-oriented functionality and syntax can be seen when the classes are address through the use "PUBLIC" statements 304. Further, on page-7, the class and instance variables are declared 306. However, as can be seen on page-9 of Figure 3 and page-8 of Figure 2, the ADN program language which was used in the pre-object-oriented and 204, is present in the object-oriented and 308. Therefore, implementing the method describe herein, the functionality and data accessibility was imported from the non-object-oriented language to an object-oriented language.

The controlling code for the object oriented ADN, which was prepared according the to method described previously, is attached hereto as Attachment A. The object-oriented ADN is a programming language for simulating the behavior of computer systems. The object-oriented ADN now supports multiple object-oriented features including classes, inheritance, constructors, method overloading, packages, interfaces and

abstract classes. The class describes a collection of data objects (i.e. constant, variables and arrays) and methods (i.e. behaviors, functions and constructors) that may use data objects. The definitions may include constants, variables, arrays, behaviors, functions and constructors. If the declaration of an object is preceded by the keyword "public", then the object is visible outside the class. Otherwise the object is only visible within the class. The class is nothing but a template. Defining a class allocates no storage to the data objects in the class. It is the instances of the class that allocate storage and manipulate the data.

A constructor is a special function within a class that is evoked automatically when a class is instantiated. The name of the constructor must be the same as the name of the class. The body of the constructor must execute in zero simulation time. Simulation time is calculated, not recorded as actual time. Simulation time is the amount of time necessary to accomplish a task if the task was being done in an actual computer system. As the software is simulating the computer system, the software can execute tasks with zero simulation time, thus that time is not included in the time analysis of the computer system. Simulation time is important because performance statistics are calculated from the simulation times at which events occur. Simulation time is maintained in a double precision floating point variable that measures the number in seconds from when the simulation begin.

A behavior is a collection of ADN code that is invoked as a procedure. Its body may take simulation time to execute. It can take parameters as inputs and can return multiple values as outputs. A variable is declared by specifying its type, its name, and an optional initial value. A constant is a variable or array whose value cannot change during

execution. A constant is created by preceding the declaration of a variable or array with the keyword "final".

A class is instantiated using the new operator. Instantiating a class creates a new instance of the class by allocating the needed storage, invokes a constructor for the class with the provided parameter values and returns a reference or pointer to newly created instance.

Within a class there may exist multiple functions or multiple behaviors having the same name, as long as each function or behavior has different number of parameters. This is called "Method Overloading". When an overloaded function or behavior is invoked, the version having the same number of parameters as values being passed is invoked. Method Overloading allows implementing several variations on a method and can be used in place of a single method having a variable number of parameters.

A package is a collection of constants, variables, arrays, behaviors, functions, classes and interfaces. A package is used to group related objects, making some of the objects visible outside of the package and hiding others within the package. A package is not instantiated. The objects in a package are referenced using their simple names. If the declaration of an object in a package is preceded by the word "public", then that object is visible outside the package. Otherwise, the object is visible only within the package. If the same name appears in multiple package directives, all the objects are considered to be a part of the same package. This allows a package to be defined in a non-contiguous pieces spanning several files.

Several key ADN features in existence before the object-oriented extension were enhanced during the conversion, adding significant value to the objected-oriented ADN. The features are as follows:

Application Logic. Originally, ADN application logic was exclusively implemented in ADN global behaviors that resemble conventional programming subroutines. Simple logic statements were provided for conditional testing, looping, and data input and output operations. Performance statements were provided to account for hardware resource usage. Simulated send and receive statements were provided for inter-process communication. A set of built-in utility functions rounds were available to behaviors.

After Object Oriented ADN, application logic could also be defined in class behaviors providing logic encapsulation, a key feature in Object Oriented ADN reusability. In addition, the user-defined Object Oriented ADN function was introduced in both global and class-contexts. The Object Oriented ADN function is limited to zero simulated time logic that can be invoked within the context of an expression evaluation. The class-context function is equivalent to the traditional Object Oriented method. The class constructor, a special function, was added for the initialization of a newly instantiated object. This can be seen through a comparison of the application logic 206 of the pre-Object Oriented ADN shown in FIGURE 2 with the application logic 310 of the Object Oriented ADN shown in FIGURE 3.

Data Types. Before Object Oriented ADN, data consisted of type-less constants, variables, and arrays that took on the type of the data currently being held. Valid data types were Integer, Real, String, and Undefined. To conserve memory, 1, 2, 4, 8, and 16

bit array elements were also available. Associative arrays provided a means of storing a data values with a string value as a subscript.

After Object Oriented ADN, the former type-less variables are given the type designation of Generic. The object reference is added to the generic type. In addition, strongly typed variables of Integer, Real, String, and class-type are added to provide the same data type capability available in traditional Object Oriented languages such as Java. Examples can be seen in the adn.y file attached as the computer program listing appendix submitted on compact disk and include the following:

```
array_init_list: expr
                        { $$ = 1; }
        | array_init_list ',' expr
                        { $$ = $1+1; }
        | array_init_list ','
                        { $$ = $1; }
            ;

data_element_type: GENERIC
                        { $$ = (Inst *)GENERIC; }
            | VARIABLE
                        { $$ = (Inst *)GENERIC; }
        | CONST
                        {
                          modif |= FINAL_mo;
                          $$ = (Inst *)CONST;
                        }
            | STROP
                        { $$ = (Inst *)STROP; }
            | REALOP
                        { $$ = (Inst *)REALOP; }
            | INTOP
                        { $$ = (Inst *)INTOP; }
        | TYPE
            { $$ = (Inst *)$1; }
        | STRING
            {
                          $$ = (Inst *)$1; data_element_token = STRING;
                        }
        | ASSOCIATIVE
```

```
                { $$ = (Inst *)ASSOCIATIVE; }
            | DATUM
                { $$ = (Inst *)DATUM; }
                | INTOP ICONST BIT
                        {
                         if ( strcmp($2,"16") == 0 ) {
                                $$ = (Inst *)INTEGER16;
                         }
                         else if ( strcmp($2,"8") == 0 ) {
                                $$ = (Inst *)INTEGER8;
                         }
                         else if ( strcmp($2,"4") == 0 ) {
                                $$ = (Inst *)INTEGER4;
                         }
                         else if ( strcmp($2,"2") == 0 ) {
                                $$ = (Inst *)INTEGER2;
                         }
                         else if ( strcmp($2,"1") == 0 ) {
                                $$ = (Inst *)INTEGER1;
                         }
                         else {
                                yyerror("Invalid Integer modifier");
                        }
                         }
                ;
data_element_type2: /* null */
                { $$ = (Inst *)GENERIC; }
            | GENERIC
                        { $$ = (Inst *)GENERIC; }
                | STROP
                        { $$ = (Inst *)STROP; }
                | REALOP
                        { $$ = (Inst *)REALOP; }
                | INTOP
                        { $$ = (Inst *)INTOP; }
            | TYPE
                { $$ = (Inst *)$1; }
            | STRING
                { $$ = (Inst *)$1; data_element_token = STRING; }
            | ASSOCIATIVE
                { $$ = (Inst *)ASSOCIATIVE; }
                | INTOP ICONST BIT
                        {
                         if ( strcmp($2,"16") == 0 ) {
                                $$ = (Inst *)INTEGER16;
                         }
```

```
        else if ( strcmp($2,"8") == 0 ) {
                $$ = (Inst *)INTEGER8;
        }
        else if ( strcmp($2,"4") == 0 ) {
                $$ = (Inst *)INTEGER4;
        }
        else if ( strcmp($2,"2") == 0 ) {
                $$ = (Inst *)INTEGER2;
        }
        else if ( strcmp($2,"1") == 0 ) {
                $$ = (Inst *)INTEGER1;
        }
        else {
                yyerror("Invalid Integer modifier");
    }
     }
;
```

Data Scope. Before Object Oriented ADN, the available data scopes consisted of model global including behavior names, simulated file, table and index names. In addition there were process local, thread local, and behavior local data scopes.

After Object Oriented ADN, the data scopes were expanded to include package scope 312, class scope 314, and object instance scope 316. These added data scopes provide the data hiding and logic encapsulation necessary in the creation of reusable model components.

Process and Thread Instances. Before Object Oriented ADN, process and thread structure instances were created and managed by internal logic. Selected state data was accessible through a set of built-in functions.

After Object Oriented ADN, classes were defined for processes and threads. Examples can be seen in the Utilities.adn file attached as the computer program listing appendix submitted on compact disk and include the following, noting that the constructors are native and private:

```
//---------------------------------
// *** process and thread support ***
//---------------------------------

// must match front end of t_Process structure definition in Software_Util
// allows any defined fields to be accessed by ADN

public final class ses_Thread {

        private integer                 reserved;
        private Integer                 object_ptr; // ptr to associated t_Object
        private                         ses_SharedMessageQueue
fSharedMessageQueue;
        private native constructor      ses_Thread(); // used internally

}

public final class ses_Process {

        private integer                 object_ptr; // ptr to associated t_Object
        private ses_Thread              fMainThread;
        private ses_SharedMessageQueue  fSharedMessageQueue;
        native private constructor      ses_Process(); // used internally

}
```

Message Instances. Before Object Oriented ADN, message structure instances were created and managed by internal logic. Selected state data was accessible through a set of built-in functions.

After Object Oriented ADN, a class was defined for the message. Through this class, message fields may be accessed by the ADN user. The server classes use this class to handle the routing of service requests to the corresponding service behavior. Instances of the message class are accessible in the operating system logic where message activity is intercepted. Examples can be seen in the Utilities.adn file attached as the computer program listing appendix submitted on compact disk and include the following, noting that the constructors are native and private:

```
//------------------------
```

```
// *** message support ***
//-----------------------

// must match t_Msg structure definition in Software_Util.Message.sim

public final Class ses_Message {

    private native constructor ses_Message(msg_ptr) Returns(msg_ref);

  public native static function associatedMsg(msg_ptr) Returns(msg_ref);
  public native function sendToHardware(target_id,Kbytes) Returns();

    private Integer object_ptr; // ptr to associated t_Object
    private Integer statStack; // t_StatStack *
    private Real send_time;
    private Real receive_time;
    private Real reply_time;

  public Real     message_bytes;
  public Datum  data[]; // struct Datum *, also int ndata

    private Integer nextMsg; // t_Msg *

  public Integer sending_proc_sn;
  public Integer receiving_proc_sn;

    private Integer client_proc_sn;
    private Integer timeout_proc_sn; // <0 means stale

  public Integer msg_protocol; // protocol

    private Integer iDbTransaction; // associated transaction of sending
proc
    private Integer msg_type; // t_MsgType type;
    private Integer 1 bit forward;
    private Integer 1 bit svcState;
    private Integer 1 bit local;
    private Integer 1 bit xfrDb;
    private Integer 2 bit passCount;

}
```

Client Workload. Before Object Oriented ADN, the logic for a client workload was generated as an ADN behavior executing under a special user process. Client

workloads initiated the execution of behavior logic on a computer. Workloads could include human factor think time or could be expressed in terms of inter-arrival time.

After Object Oriented ADN, the default functionality remains unchanged. By adding state data and additional logic, a user can define significantly more complex workload models. For example, the public factory behavior 318 of FIGURE 3 is shown. A client workload object can be created in a similar manner.

Server Process. Before Object Oriented ADN, server processes were specified through a graphical user interface (GUI) which in turn generated ADN behaviors according to internally defined patterns and user supplied information. Services are specified by ADN behaviors supplied by the modeler.

After Object Oriented ADN, server process infrastructure was designed and implemented as ADN classes. The new server infrastructure can be seen in server.adn attached as the computer program listing appendix submitted on compact disk

Database Server. Before Object Oriented ADN, a built-in Oracle model provided the only significant database modeling capability. This feature allowed for the definition of cache size, block size, and individual named tables and indexes sizes. The semantics for a database transaction were supported in terms of starting and committing a transaction and collecting response statistics per transaction type. A transaction could be defined in the logic of a behavior to use select, insert, update, and delete operations on table rows. Operations such as update could optionally simulate row level locking. Distributed, central, and parallel Oracle model functions were supported.

After Object Oriented ADN, users can create models that involve significant data modeling. An example of the database server modeling capability can be seen in dbms.adn attached as the computer program listing appendix submitted on compact disk.

Operating System. Before Object Oriented ADN, a default parameterizeable operating system model was supplied with the product. A user could create a modified operating system model although there was a substantial risk that the integrity of the model infrastructure would be compromised.

After Object Oriented ADN, the operating system model logic was encapsulated in an operating system class. With this new design, a user can safely extend the operating system without disturbing the original logic. Examples include the base operating system functionality 320 of FIGURE 3 and a user defined operating system extension shown in user_extensions.adn attached as the computer program listing appendix submitted on compact disk.

Statistics. Before Object Oriented ADN, all statistics were built-in. The user was able to control which statistics would be collected and reported.

After Object Oriented ADN, a user definable statistics capability was implemented as a set of Object Oriented ADN classes. A user defined query of built-in statistics during simulation was also added. Examples can be seen in the Utilities.adn file attached as the computer program listing appendix submitted on compact disk and include the following:

```
public class ses_Statistic {

    // constants
    static final integer    discrete = 0;
    static final integer    continuous = 1;
    static final integer    delta = 0;
```

```
static final integer     absolute = 1;

// state fields (must match t_Statistic in ADNparser.Statistic
private integer          fHandle;
private integer          fType;
private string           fName;
private integer          fNpercentiles; // for liveDistribution

// interface to Workbench statistic accessor methods
// ------------------------------------------------

public native function count() returns(number_samples);
public native function deviation() returns(deviation);
public native function duration() returns(duration);
public native function maximum() returns(maximum_value);
public native function minimum() returns(minimum_value);
public native function mean() returns(mean_value);
public native function name() returns(name_string);
public native function reporting(state) returns();
public native function sample(value) returns();
public native function sampleAbsolute(value) returns();
public native function type() returns(type_string);
public native function value() returns(last_sample);
public native function variance() returns(variance);

private native Function defineDiscrete(prefix) returns();
private native Function defineContinuous(prefix) returns();
private native Function registerQueryStatistic() returns();
public  native Function isActive() returns();

public native function registerService(serviceRef,serviceName) returns();

// constructors
// ------------

// used by ses_StatisticsManager.createQueryStatistic
// note: locateStatistic sets statType and statHandle

public Constructor ses_Statistic( aName ) {
        fName = aName;
        fType = 2;      // query stat type
        this.registerQueryStatistic();
}
public Constructor ses_Statistic( aName, aIntervalWidth ) {
        fName = aName;
        fType = discrete;
```

```
}

public function createIntervalStatistic( aName, aIntervalWidth ) {
        ses_Statistic tIntervalStat;
        generic tStatMgr = ses_gStatMgr; // overcome lookahead problem
        tIntervalStat = new ses_Statistic( aName, aIntervalWidth );
        this.add_interval( "#UDD%d#" + aName, aIntervalWidth, tIntervalStat );
        tStatMgr.fDiscreteStatisticGroup.insertStatistic( tIntervalStat );
        return( tIntervalStat );
}

// used by ses_StatisticsManager.createDiscreteStatistic and
//         ses_StatisticsManager.createContinuousStatistic

public Constructor ses_Statistic( aUxxPrefix, aName, aType  ) {
fName = aName;
fType = aType;
switch ( fType ) {
                case( discrete ) {
                this.defineDiscrete( aUxxPrefix );
                }
                case( continuous ) {
                this.defineContinuous( aUxxPrefix );
                }
                default {
                Error "Invalid statistic type, must be discrete or continuous.";
                }
}
}
```

Utility Classes. Before Object Oriented ADN, there were no general-purpose utility classes.

After Object Oriented ADN, several utility classes were added, including the following: a list class for handling lists of objects; a list iterator class for accessing individual objects in a list; a semaphore class for use in controlling access to data; and logic that can only be processed by one thread at a time. Examples can be seen in the Utilities.adn file attached as the computer program listing appendix submitted on compact disk.

Garbage Collection. Before Object Oriented ADN, memory management of all dynamically created structures such as processes, threads, and messages were built-into the modeling system.

After Object Oriented ADN, memory management of the objects can be explicitly instantiated by a user. A system of object reference counts and automatic garbage collection were implemented based on the Java language semantics. The following functions were included and can be seen in ADNparser30.sim attached as the computer program listing appendix submitted on compact disk: init_gc_proc, incr_ref_count, decr_ref_count, and object_destroy.

Many of the advancements of the Object Oriented ADN are disclosed herein, however, the Object Oriented ADN code is attached as the computer program listing appendix submitted on compact disk, which is hereby incorporated by reference.

The foregoing disclosure and description of the invention are illustrative and explanatory thereof and various changes to the size, shape, materials, components, and order may be made without departing from the spirit of the invention.